# TP3 : Classe et objet

Dans ce TP, nous allons écrire un programme qui va trier des tableaux de nombres complexes.

## Exercice 1 : Définition du type nombre complexe

a. Définissez le type `NbComplexe` à l'aide d'une classe. Rappel : un nombre complexe est défini par sa partie réelle (que vous pourrez appeler `re`) et sa partie imaginaire (que vous pourrez appeler `im`).

```
class NbComplexe {
  public:
    float re, im;
};
```

b. Ajoutez deux procédures membres à cette classe : une pour saisir le nombre complexe au clavier (initialisation des données membres) et une pour afficher le nombre complexe à l'écran sous le format : `re +im i` si la partie imaginaire est positive ou nulle, et `re -im i` si elle est négative.

```
void saisir() {
  cout << endl << "Saisir la partie reelle: ";
  cin >> re;
  cout << "Saisir la partie imaginaire: ";
  cin >> im;
}

void afficher() const {
  cout << re << " ";
  if (im >= 0.0) cout << "+";
  cout << im << " i";
  cout << endl;
}
```

c. Créez un main qui crée un nombre complexe sur la pile, l'affiche, puis le saisi et le réaffiche. Qu'obtenez-vous ?

```
int main () {
  NbComplexe unComplexe;
  unComplexe.afficher();
  unComplexe.saisir();
  unComplexe.afficher();
  return 0;
}
```

A la déclaration du nombre complexe, le constructeur par défaut est appellé, les deux données ont des valeurs indéterminées (pas forcément zéro), la saisie affecte les valeurs de l'utilisateur aux données qui sont ensuite celles affichées.

d. Quelle instruction permet d'afficher la taille, en octets, d'un nombre complexe ?

```
cout << sizeof(NbComplexe);
// ou : cout << sizeof(unComplexe); avec unComplexe objet de type NbComplexe
```

NB: L'espace mémoire total requis pour un objet est la somme des espaces nécessaires pour chaque donnée membre, plus un **padding interne optionel**. Quand un ordinateur moderne lit ou écrit vers une addresse mémoire, il le fait sur des mots. Un alignement des données signifie mettre les données à une adresse décalée égale à un multiple de la taille d'un mot, ce qui augmente les performances du système de

part la façon dont le CPU gère la mémoire. Pour aligner les données, il peut donc être nécessaire d'insérer des octets (non significatifs) entre les données membres d'un objet.

## Exercice 2 : Constructeur, destructeur et allocation dynamique

a. Ajoutez à la classe `NbComplexe` trois constructeurs et un destructeur. Le premier constructeur sera sans paramètre, le deuxième aura deux paramètres pour les deux parties du nombre complexe, et le troisième est un constructeur par copie.

```
NbComplexe () : re(0), im(0) { }

NbComplexe (float a, float b) : re(a), im(b) { }

NbComplexe (const NbComplexe& c) {re = c.re; im = c.im;}

~NbComplexe () { }
```

b. Ajoutez une procédure membre `multiplier` qui multiplie le nombre complexe par un autre nombre complexe passé en paramètre. L'instance courante contient le résultat de la multiplication, le nombre complexe avec lequel on multiplie n'est pas modifié.

Rappel : $(re_1 + im_1\, i) \times (re_2 + im_2\, i) = (re_1 \times re_2 - im_1 \times im_2) + (im_1 \times re_2 + re_1 \times im_2)\, i$

```
void multiplier (const NbComplexe& c){
  float saveRe = re;
  re = re*c.re - im*c.im;
  im = im*c.re + saveRe*c.im;
}
```

c. Complétez le main en ajoutant la création d'un nouveau nombre complexe comme étant une copie du nombre saisi à l'exercice précédent (affichez le pour vérifier que votre constructeur fonctionne correctement). Ensuite créez un nombre complexe différent **sur le tas** et affichez le. Finallement, multipliez les deux nombres et affichez le résultat.

```
NbComplexe complexe1 (unComplexe);
complexe1.afficher();
NbComplexe * complexe2 = new NbComplexe(5,2);
complexe2->afficher();
complexe2->multiplier(complexe1);
complexe2->afficher();
delete complexe2;
```

## Exercice 3 : Comparaison de deux nombres complexes

Ajoutez dans la classe `NbComplexe` les deux fonctions membres suivantes :

- une fonction `module` qui retourne le module du nombre complexe. Rappel : $|re + im\, i| = \sqrt{re^2 + im^2}$
- une fonction `estPlusPetit` qui indique si le nombre complexe est plus petit qu'un autre nombre complexe passé en paramètre. La comparaison se fera sur les valeurs des modules des deux nombres.

Testez ces deux fonctions sur quelques nombres complexes.

```
float module() const {
  return sqrt(re*re+im*im);
}

bool estPlusPetit (const NbComplexe & c) const {
```

```
  return module() < c.module();
}
```

## Exercice 4 : Création d'un tableau de nombres complexes aléatoires

a. Dans le main, allouez de la mémoire pour un tableau de nombres complexes dont la taille sera saisie par l'utilisateur.

b. Remplissez le tableau avec des nombres complexes dont les parties réelles et imaginaires sont tirées aléatoirement dans l'intervalle [-10,10] avec exactement 1 décimale de précision. L'annexe B décrit comment faire un tirage aléatoire d'une valeur entière dans un intervalle.

c. Affichez le tableau de nombres complexes ainsi rempli et pour chaque élément du tableau affichez également le module du nombre complexe. N'oubliez pas de libérer la mémoire allouée dynamiquement quand vous en n'avez plus besoin.

```
int taille;
cout << "Donner le nombre de complexes: ";
cin >> taille;
if (taille <= 0) return 0;
srand((unsigned int) time(NULL));
NbComplexe * tab = new NbComplexe[taille];
for (int i=0; i<taille; i++) {
  tab[i].re = ((rand()%201)/10.0)-10.0;
  tab[i].im = ((rand()%201)/10.0)-10.0;
  cout << "tab[" << i << "]= ";
  cout << "(mod= " << tab[i].module() << ") ";
  tab[i].afficher();
}
delete [] tab;
```

## Exercice 5 : Tri par sélection du tableau de nombres complexes

Définissez une procédure globale (pas une procédure membre) `trierParSelection` qui prend en paramètres un tableau de nombres complexes et sa taille, et qui le trie du nombre le plus petit au plus grand (en termes de module), en utilisant l'algorithme de tri par sélection. Testez votre procédure dans le main.

```
void trierParSelection (NbComplexe * tab, int taille) {
  NbComplexe minComplexe;
  for (int i=0; i<taille-1; i++) {
    int indmin = i;
    for (int j=i+1; j<taille; j++) {
      if (tab[j].estPlusPetit(tab[indmin])) indmin = j;
    }
    minComplexe = tab[indmin];
    tab[indmin] = tab[i];
    tab[i] = minComplexe;
  }
}

int main () {
  int taille;
  cout << "Donner le nombre de complexes: ";
  cin >> taille;
  srand((unsigned int) time(NULL));
  if (taille <= 0) return 0;
  NbComplexe * tab = new NbComplexe[taille];
  for (int i=0; i<taille; i++) {
    tab[i].re = ((rand()%200)/10.0)-10.0;
```

```
    tab[i].im = ((rand()%200)/10.0)-10.0;
    cout << "tab[" << i << "]= ";
    cout << "(mod= " << tab[i].module() << ") ";
    tab[i].afficher();
  }

  trierParSelection(tab,taille);
  for (int i=0; i<taille; i++) {
    cout << "tab trie[" << i << "]= ";
    cout << "(mod= " << tab[i].module() << ") ";
    tab[i].afficher();
  }
  delete [] tab;
  return 0;
}
```

On a le droit de faire les affections entre nombres complexes car les données membres sont deux réels, on n'a donc pas besoin de définir l'opérateur d'affectation ici.

## Exercice 6 : Tri par insertion du tableau de nombres complexes

Définissez une procédure globale (pas une procédure membre) `trierParInsertion` qui prend en paramètres un tableau de nombres complexes et sa taille, et qui le trie du nombre le plus petit au plus grand (en termes de module), en utilisant l'algorithme de tri par insertion. Testez votre procédure dans le main.

```
void trierParInsertion (NbComplexe * tab, int taille) {
  NbComplexe complexeAPlacer;
  for (int i=1; i<taille; i++) {
    complexeAPlacer = tab[i];
    int j = i - 1;
    while ( j >= 0 && complexeAPlacer.estPlusPetit(tab[j]) ) {
      tab[j+1] = tab[j];
      j--;
    }
    tab[j+1]=complexeAPlacer;
  }
}
```

Le programme principal est identique à celui de la question précédente en changeant uniquement l'appel à la procédure de tri, ici : `trierParInsertion(tab,taille);`